VŠB - Technická univerzita Ostrava Fakulta elektrotechniky a informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2018 Jakub Dolníček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Dolníček**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Webvalley s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. RNDr. Petr Šaloun, Ph.D.**

Konzultant bakalářské práce: Bc. Lukáš Vlček

Datum zadání: 01.09.2017
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2018 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 20. dubna 2018

.....

webvalley Webvalley s.r.o.
Adresa: Pohraniční 504/27 703 00 Ostrava
Kontakt: info@webvalley.cz www.webvalley.cz
IČO: 29 44 85 31 DIČ: CZ 29 44 85 31

Rád bych poděkoval všem zaměstnancům firmy Webvalley s. r. o. za přátelské pracovní prostředí a odbornou pomoc. Speciálně pak Mgr. Tomáši Jančarovi a Ing. Martinu Drozdkovi za jejich nezlomnou trpělivost a nápomocnou ruku.

Abstrakt

V rámci mé bakalářské praxe ve firmě Webvalley s.r.o. jsem řešil více dílčích úkolů na více projektech. Tento dokument obsahuje popis splněných úkolů během mé praxe a postup jejich řešení. Hlavními úkoly bylo testování aplikace pomocí funkcionálních a akceptačních testů, migrace dat ze staré webové aplikace do nové v uživatelsky přívětivém prostředí.

Klíčová slova: PHPUnit, Codeception, PHP, Symfony, Docker

Abstract

As part of my bachelor's degree in Webvalley s.r.o. I have solved more partial tasks on multiple projects. This document describes the accomplished tasks during my practice and how to deal with them. The main tasks were to test the application using functional and acceptance tests, to migrate data from an old web application to a new one in a user-friendly environment.

Key words: PHPUnit, Codeception, PHP, Symfony, Docker

Obsah

Seznam použitých zkratek a symbolů

CRM
- Customer Relationship Management

PHP
- Hypertext Preprocessor

ORM
- Objektově relační mapování

IDE
- Integrated development environment

PSR
- PHP Standards Recommendations

PHP-FIG
- Framework Interop Group

MVC
- Model view controller

Doctrine 2
- PHP knihovna pro ORM

ElasticSearch
- Knihovna pro fulltextové vyhledávání

URL
- Uniform Resource Locator je řetězec znaků s definovanou strukturou, který slouží k přesné specifikaci umístění zdrojů informací na Internetu.

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V této práci popisuji mou odbornou praxi ve společnosti Webvalley s. r. o. Rozhodl jsem se ji absolvovat, jelikož velké množství zaměstnavatelů v IT klade stále větší důraz kromě teoretických znalostí nabytých v rámci studia také na praktické zkušenosti, kde se musí setkávat s reálnými problémy a prací pod tlakem. Odborná praxe je jedna z možností, jak si tyto dovednosti osvojit a zúročit své teoretické znalosti v praxi. Webvalley s.r.o. jsem si zvolil, jelikož mě zaujmuli technologie, na kterých své projekty staví na moderních technologiích a chtěl jsem se s nimi touto formou více seznámit.
V této bakalářské práci Vás seznámím náplní mé bakalářské praxe a popíšu jednotlivé úkoly, na kterých jsem pracoval.

2. Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta

2.1 Popis odborného zaměření firmy

Firma Webvalley s. r. o. byla založena 26.6.2012 a zabývá se poskytováním softwarových řešení na míru jako webový vývoj, uživatelské testování, mobilní aplikace, optimalizace a přízpůsobení jednotlivých softwarových produktů pro účel zákazníkových požadavků jako A/B testování a analytika. Firma sídlí ve Vítkovicích v Ostravě.

2.2 Pracovní zařazení studenta

V rámci mé odborné praxe jsem pracoval jako PHP programátor. Náplň mé práce ve firmě byl převážně vývoj e-shopu pro TONAK a. s. jež je firma zabývající se výrobou klobouků a vývoj CMR systému Necktie, který slouží k správě uživatelů.

3 Seznam úkolů zadaných studentovi v průběhu odborné praxe

3.1 Seznámení s vývojářskými nástroji a standarty

Ve firmě jsme pracovali s vývojářským nástrojem PhpStorm s kterým jsem se seznámil. Dále jsem musel nastudovat dokumentaci populárního PHP frameworku Symfony, v kterém firma většinu svých projektů vyvíjí. Proto jsem dostal jako první úkoly seznámení se s tímto frameworkem a splnění běžných úkolů jako je chápání jeho struktury, práce s formuláři, ORM atd. Jelikož se správnost odvedené práce ověřovala testy. Naučil jsem se pracovat také s PHPUnit a později i s testovacím frameworkem Codeception jež je rozšířením nad zmiňovaným PHPUnit.

3.2 Testování Symfony bundlů

Pro systém Necktie byly vyvíjeny podpůrné bundly, řešící určitou funkcionalitu v rámci širšího kontextu projektu. Tyto bundly bylo nutné otestovat pomocí Unit testů. Pro zautomatizovaní testování správnosti funkcionality při změnách což šetří programátorský čas.

3.3 Tvorba akceptačních testu pro případy užití v rámci systému Necktie

Úkolem bylo vytvoření akceptačních testů, které prověřují jednotlivá reálná použití Necktie systému. Vytvořením těchto testů zaručuje kontrolu, zda při průběhu nějakého případu užití nenastane chyba a zda na konci dostaneme očekávaný výsledek.

3.4 Vytvoření importu dat do nového systému skrze CSV soubory

V tomto úkolu bylo požadováno z přijatých dat ze starého systému vytvořit nové databázové záznamy splňující požadavky na logiku nového systému a udržet zároveň všechny logické vazby mezi jednotlivými daty.

4. Zvolený postup řešení

4.1 Seznámení s vývojářskými nástroji a standarty

Pro vývoj jsme využívali IDE PhpStorm, který programátorovi ulehčuje hodně opakující se práce jako generování getrů a setrů. Pomáhá dodržovat PHP standarty PSR. PSR je sada pravidel pro přehlednost a udržovatelnost kódu určená skupinou PHP-FIG. Dálé jsem se naučil pracovat s PHP Frameworkem Symfony, což je framework založený na architektuře MVC. Díky strukturování logických částí do bundlů v rámci Symfony frameworku, vytváří možnost znovupoužitelnosti již vytvořených částí aplikaci. Programátor tak může využít již hotových vlastních řešení určitých funkcionalit, nebo využít řešení třetích stran díky správci závislostí Composeru. Programátor tak má více prostoru na řešení aplikace. Testovat jsem se nejdříve naučil v PHPUnit, který umožňuje unit testy. Poté jsem své dovednosti rozšířil o testovací framework Codeception, který oproti PHPUnit umožňuje také funkcionální a akceptační testy.

4.2 Testování Symfony bundlů

V rámci tohoto úkolu jsem testoval dva bundly trinity-setting sloužící k ukládání Symfony parametrů do databáze za pomocí ORM Doctrine 2 a ulehčuje tak přístup k těmto parametrům z kontroleru a šablony. Druhý testovaný bundle byl trinity-logger, který obstarává log z ElasticSearch databáze. Pro testy jsem využil testovacího frameworku PHPUnit.

Oba bundly jsou vyvíjeny v Docker kontejnerech, což zajištuje stejné prostředí apache a mysql na lokálních strojích jako je na produkčním hosting, tím se zamezí rozdílnému chovaní mezi vývojem a produkci.

Na těchto projektech jsme využili Docker Community Edition a Docker Compose, který je nádstavbou umožňující zavedení kontejnerů pomoci konfiguračního souboru Dockerfile kde se specifikují nastavení jednotlivých kontejnerů sloužících jako vývojové prostředí.

Při testovaní je dobrým standardem dodržovat strukturu testů stejnou jako je původní struktura projektu a testy nazývat stejně jako jednotlivé testované celky.

Pro tvorbu PHPUnit testů potřebujeme nejdříve danou knihovnu stáhnout do našeho projektu. Pro tento účel jsme v projektech využívali správce závislostí Composer. Pro stažení PHPUnit jsem musel definovat název tohoto balíčku i s požadovanou verzí v souboru composer.json, kde se specifikují požadované závislosti. Tyto závislosti se pak dále dělí na dvě hlavní skupiny produkční a vývojové kam patří i PHPUnit, jelikož na produkci nemá tento balíček žádný význam.

```
"require": {
    "php": ">=7.1",
    "doctrine/orm": "~2.5",
    "symfony/http-kernel": "~2.8|~3.0",
    "symfony/dependency-injection": "~2.8|~3.0",
    "symfony/config": "~2.8|~3.0",
    "twig/twig": "~1.20|~2.0",
    "twig/extensions": "~1.3"
  },

  "require-dev": {
      "doctrine/doctrine-bundle": "~1.6",
      "symfony/var-dumper": "~3.0",
      "symfony/browser-kit": "~2.8|~3.0",
      "symfony/form": "~3",
      "symfony/validator": "~3",
      "symfony/twig-bundle": "~2.8|~3.0",
      "mockery/mockery": "~0.9",
      "fzaninotto/faker": "~1.5",
      "satooshi/php-coveralls" : "~1",
      "phpunit/phpunit": "~6",
      "phpstan/phpstan": "~0.7"
  },
```

Výpis 1: Ukázka configurace v composer.json

Jednotlivé testy se píší pomocí tříd, které dědí z PHPUnit\Framework\TestCase, což je třída poskytující hlavní funcionalitu testů. Díky této třídě získáme sadu metod, které využíváme při testování. Já jsem z této třídy dědil pro svojí abstraktní třídu BaseTest, jelikož jsem potřeboval zavést metody, které jsem potřeboval napříč všemi testy a takto jsem je mohl definovat pouze na jednom místě a získat je při dědění zároveň s funkcionalitou PHPUnit\Framework\TestCase

```php
<?php

namespace Trinity\Bundle\SettingsBundle\Tests;

use Doctrine\Bundle\DoctrineBundle\Registry;
use PHPUnit\Framework\TestCase;
use Doctrine\Common\Persistence\ObjectManager;

/**
 * Class BaseTest
 * @package Trinity\Bundle\SettingsBundle\Tests
 */
abstract class BaseTest extends TestCase
{

    /**
     * @return \PHPUnit_Framework_MockObject_MockObject
     */
    protected function getRegistry($settingsRepository): \PHPUnit_Framework_MockObject_MockObject
    {
        $registry = $this
            ->getMockBuilder(Registry::class)
            ->disableOriginalConstructor()
            ->getMock();

        $entityManager = $this
            ->getMockBuilder(ObjectManager::class)
            ->disableOriginalConstructor()
            ->getMock();

        $entityManager->method('getRepository')
            ->willReturn($settingsRepository);

        $registry->method('getManager')
            ->will(static::returnValue($entityManager));

        return $registry;
    }


    /**
     * Return user id. $this->getUser()->getId();
     * @return int
     */
    protected function getUserId(): int
    {
        return 1;
    }
}
```

Výpis 2: Implementace BaseTest třídy

Jelikož jsem měl testovat funkcionalitu Symfony bundlů, které bez zavedení do jiného projektů nemají žádná data, které by testovala jejich funkcionalitu. Rozhodl jsem se, že je budu testovat pomocí mocků. Mockování je technika, která vytvoří požadovaný objekt, který jinak za daných okolností nemáme k dispozici v našem případě je v kostře aplikace, kam bundle v budoucnosti zavedeme nebo mockem simulujeme návratové hodnoty po dotazech z databáze atd... , ale potřebujeme ho k otestovaní dané funkcionality.

Pro mockování jsem využíval metod přímo knihovny PHPUnit, který tuto techniku podporuje.

```
1.  $owner = $this->getMockBuilder(Owner::class)
2.              ->disableOriginalConstructor()
3.              ->setMethods(['getId'])
4.              ->getMock();
5.
6.  $owner->expects(static::any())
7.              ->method('getId')
8.              ->will(static::returnValue(1));
```

Výpis 3: Ukázka mockování

Do metody getMockBuilder vložíme jako parametr třídu, kterou chceme mockovat.
Jednotlivé metody MockBuilderu vždy provedou nějakou funkcionalitu, a poté vrátí opět instanci samotného MockBuilderu, což nám umožnuje jednotlivé metody řetězit tak jak je vidět v ukázce.
Dále jsem v tomto případě deaktivoval konstruktor, jelikož okolnosti neumožnovali vložit potřebné parametry konstruktoru. Metodou setMethods se skrze parametr definuje, které metodě chceme nastavit falešnou funkcionalitu. Poté na instanci našeho mocku voláme například expect(), čímž můžeme definovat kolikrát může být daná falešná metoda testována např. static::once() bude volána pouze jedenkrát, static::exactly(6) právě šestkrát a další metody specifikující nejvýše a nejméně X-krát atd. Pokud tyto podmínky nejsou dodrženy test se vyhodnotí jako neúspěšný. Dále definujeme pro kterou metodu nastavujeme právě návratovou hodnotu, popřípadě můžeme podmínit a určit při různých parametrech metody různé návratové hodnoty pomocí metody with(), kde parametrem je pole s definovanými podmiňujícími parametry. Metodou will() nastavíme návratovou hodnotu, kterou může být jakýkoliv datový typ nebo objekt. S takto vytvořeným mockem dále pracujeme stejně jako s normální instancí třídy, kterou právě mockujeme. Funkcionalitu jednotlivých metod tříd jsem testoval pomocí assertů, což jsou metody co očekávájí specifickou návratovou hodnotu nebo datový typ atd. a jako parameter vkládáme právě metodu, čímž dosadí návratovou hodnotu do assertu a tím se ověří, zda vše funguje jak očekáváme. Nejčastěji jsem využil.

- assertTrue – hodnota je datového typu boolean o hodnotě TRUE
- assertFalse – hodnota je datového typu boolean o hodnotě FALSE
- assertNull – hodnota je datový typ NULL
- assertEmpty – hodnota je prázdná
- assertNotEmpty – může být cokoliv, ale nesmí být prázdné
- assertInstanceOf – je instancí specifického objektu
- assertEquals - specifikujeme jakou hodnotu přesně očekáváme

Pokud chceme testovat několikrát stejnou funkcionalitu, ale pouze s jinými daty můžeme využít dataProvider, který toto umožnuje a ušetří mnoho času a udrží kód přehlednějším. Pro využití dataProvideru definujeme název metody sloužící jako data provider, který obsahuje pole parametrů pro testovací metodu a PHPUnit spustí test tolikrát, kolik dataProvider poskytuje sad parametrů pro test. Tím můžeme kód jednoduše otestovat s různými daty.

```
1.  /**
2.       * @dataProvider settingProvider
3.       *
4.       * @param $name
5.       * @param $value
6.       * @param null $owner
7.       * @param null $group
8.       */
9.      public function testSetWeirdData($name, $value, $owner = null, $group = null)
10.     {
11.         // Mocking
12.         $settingsEntity = new Setting();
13.         $settingsEntity->setName($name);
14.         $settingsEntity->setValue($value);
15.         !$owner ?: $settingsEntity->setOwner($owner);
16.         !$group ?: $settingsEntity->setGroup($group);
17.
18.         $settingsRepository = $this
19.             ->getMockBuilder(EntityRepository::class)
20.             ->disableOriginalConstructor()
21.             ->getMock();
22.
23.         $settingsRepository
24.             ->method('findOneBy')
25.             ->will(static::returnValue($settingsEntity));
26.
27.         $settings = new SettingsManager(
28.             $this->getRegistry($settingsRepository),
29.             ['parameter' => 'default']
30.         );
31.
32.         $settings->set($name, $value, $owner, $group);
33.
34.         //tests
35.
36.         /* conditions for settingProvider count of arguments */
37.         if (isset($group)) {
38.             static::assertEquals($value, $settings->get($name, $owner, $group));
39.
40.         } elseif (isset($owner)) {
41.             static::assertEquals($value, $settings->get($name, $owner));
42.         } else {
43.             static::assertEquals($value, $settings->get($name));
44.         }
45.     }
```

Výpis 4: Test s použitím data provideru

```php
/**
     * @return array
     */
    public function settingProvider(): array
    {
        return [
            ['parameter1', '1 + 2', 1, 'testingGroup'],
            ['parameter2', 'value', 987654, 'testingGroup'],
            ['parameter3', '"z\\/*/\\??$$#&&"', 2345, 'testingGroup'],
            ['parameter4', 'gdfgd', 52345234, 'testingGroup'],
            ['parameter5', 'sgd', 53425324, 'testingGroup'],
            ['parameter6', 'gsdg', 3453453 , null],
            ['parameter7', 'gsdf', null, null]
        ];
    }
```

Výpis 5: Ukázka data provideru

PHPUnit testy se spouští pomocí konzolového příkazu a vypíše jak testy dopadly.

```
$ phpunit --coverage-clover build/logs/clover.xml
PHPUnit 6.2.3 by Sebastian Bergmann and contributors.

.....................................                             40 / 40 (100%)

Time: 4.94 seconds, Memory: 18.00MB

OK (40 tests, 135 assertions)

Generating code coverage report in Clover XML format ... done

Generating code coverage report in HTML format ... done
```

Obrázek 1: Přiklad úspěšného projití testů

```
$ phpunit --coverage-clover build/logs/clover.xml
PHPUnit 6.0.9 by Sebastian Bergmann and contributors.

.....EEEEE.........EEEEEEEEF                                     28 / 28 (100%)

Time: 2.25 seconds, Memory: 10.00MB

There were 13 errors:

1) Tests\SettingsTest::testGetValue
Error: Call to undefined function Doctrine\Common\Cache\apcu_fetch()

/home/travis/build/modpreneur/trinity-settings/vendor/doctrine/cache/lib/Doctrine/Common/Cache/ApcuCache.php:36
/home/travis/build/modpreneur/trinity-settings/vendor/doctrine/cache/lib/Doctrine/Common/Cache/CacheProvider.php:212
/home/travis/build/modpreneur/trinity-settings/vendor/doctrine/cache/lib/Doctrine/Common/Cache/CacheProvider.php:185
/home/travis/build/modpreneur/trinity-settings/vendor/doctrine/cache/lib/Doctrine/Common/Cache/CacheProvider.php:78
/home/travis/build/modpreneur/trinity-settings/Manager/SettingsManager.php:277
/home/travis/build/modpreneur/trinity-settings/Manager/SettingsManager.php:153
/home/travis/build/modpreneur/trinity-settings/Manager/SettingsManager.php:84
/home/travis/build/modpreneur/trinity-settings/Tests/SettingsTest.php:214
```

Obrázek 2: Příklad neuspěšného projití testů

4.3 Tvorba akceptačních testu pro Use case v rámci systému Necktie

Na projektu Necktie jsem dostal úkol, udělat akceptační testy na problémové uživatelské scénáře. Akceptační testy v principu testují stránku stejně, jako když to dělá běžný uživatel. Proklikávají se skrze stránky, vyplňují formuláře a očekávají, že se jim něco objeví atd. s tím rozdílem, že v akceptačních testech definujeme kam se klikne a co se má zobrazit pomocí kódu. Pro tento účel jsem využíval testovací framework Codeception, který umožnuje psát unit testy, akceptační testy a funkcionální testy. Já jsem především v rámci tohoto úkolu využil akceptačních testů. Pro data vyplňované do formulářů jsem využil Faker Gemerator, což je knihovna sloužící ke generování náhodných dat v určitém formátu, který právě potřebujeme. Obsahuje sadu předdefinovaných funkcí pro generování dat pro jméno, stát, psč, adresu atd., ale můžeme si definovat i vlastní pravidlo. Pro práci s oběma knihovnami jsem je nejdříve musel zadefinovat do závislostí v Composeru stejně jako PHPUnit v části 4.2. Pro tvorbu akceptačních testů jsem si vytvořil abstraktní třídu BaseAcceptanceTestCase ve které jsem získával instanci Faker Generátoru a měl jsem ho díky tomu dostupný ve všech akceptačních testech, jelikož jsem v nich vždy z této třídy dědil. Jednotlivé testovací třídy jsem si rozdělil podle částí webů, které testovali a jejich metody pak byli jednotlivé uživatelské scénáře v těchto částech. Do těchto metod jsem jako parametr vkládal třídu Actor, což je třída obsahující metody akceptačních testu. V testu jsem nejdříve přešel na požadovanou url adresu, kde se nacházela testovaná funkcionalita a tam jsem například vyplnil formulář daty z Facker Generátoru a klikl na tlačítko s uložením a následně zkontroloval, zda se vše projevilo v databázi jak se očekává.

Nejčastěji jsem využíval následující metody třídy Actor.

- amOnPage() - parametrem je požadovaná url adresa kam chceme v akceptačním testu přejít
- waitForElementVisible() parametrem je selektor HTML elementu, který očekáváme, že se má na stránce zobrazit. Např. Po testu registrace očekáváme, že se nám objeví element se zprávou o úspěšné registraci.
- click() – tato metoda klikne na element podle HTML selektoru předaného v parametru

- fillField() – při testování formulářů využijeme k vyplnění polí daty. Já jsem jako data využíval náhodně vygenerovaná data z Faker generátoru.
- selectOption() - kde první parametr je selektor HTML inputu typu option a druhý je pak value hodnota zvoleného selektu
- seeInDatabase() - tato metoda např. ověří, že se uložilo do databáze to co jsem odeslal ve formuláři. Prvním parametrem je tabulka, ve které hodnoty hledáme a druhým je pole kde klíče specifikují název sloupce a očekávanou hodnotu v tom to sloupci. Cele se to v metodě poskládá do SQL selectu kde hodnoty předané v poli podmiňují select klauzilí WHERE.

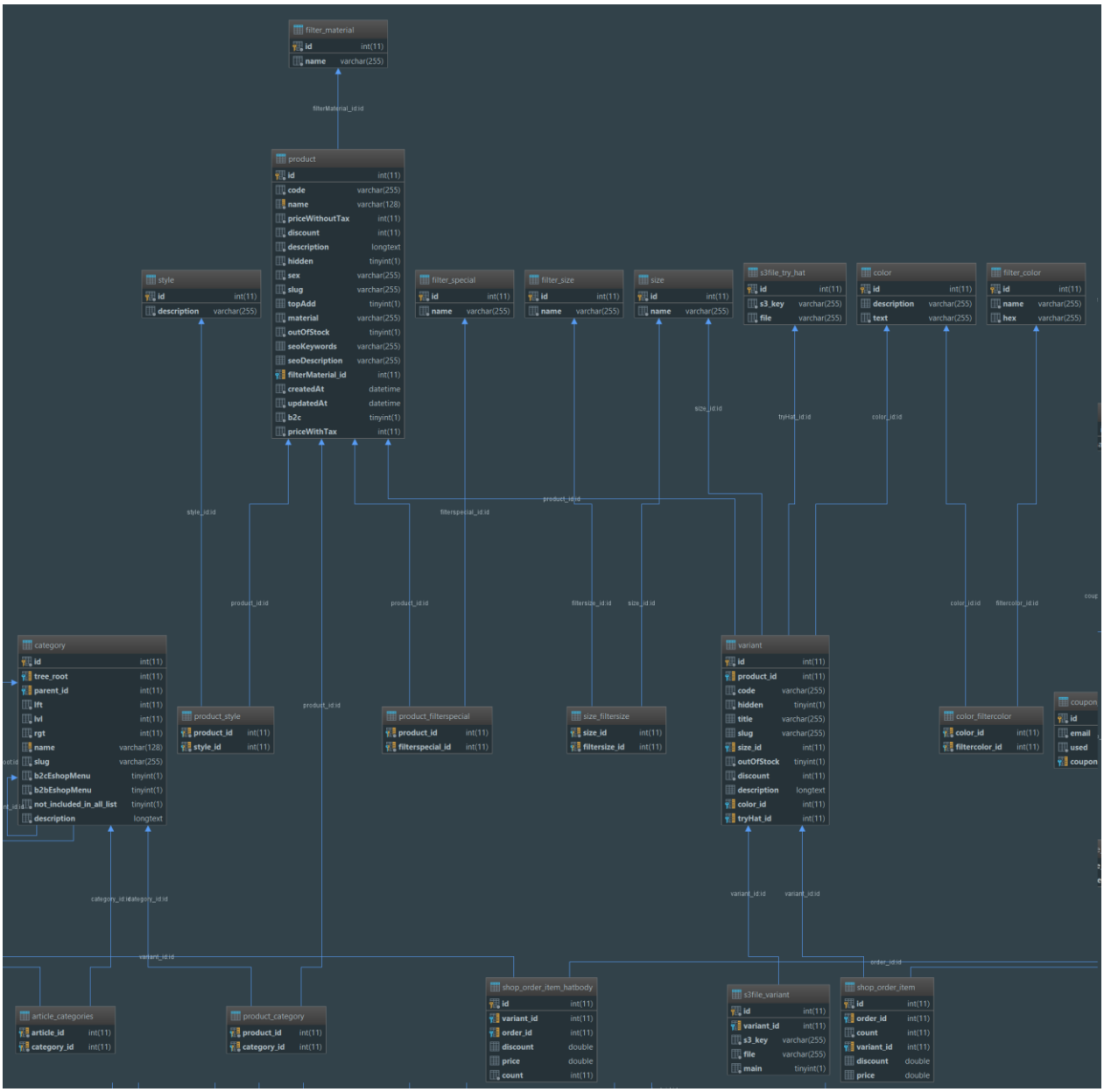
Při akceptačních testech se vyplatí často prováděné případy užití, při testech zaobalit do metod a využívat je na více místech pro udržení přehlednosti, hezčí struktury kódu a zjednodušení případných změn. Takovým typickým příkladem pro akceptační testy je přihlášení do systému. Přihlášení potřebujeme ve všech testech testujících části vyžadující autorizaci.

```
1.     /**
2.      * Login into system.
3.      *
4.      * @param string $username
5.      * @param string $password
6.      *
7.      * @throws \Exception
8.      */
9.     public function login(string $username, string $password): void
10.    {
11.        $this->amOnPage('/logout');
12.        $this->waitPageLoad(1);
13.
14.        try {
15.            $text = $this->grabTextFrom('.page-headline');
16.        } catch (\Exception $exception) {
17.            $text = null;
18.        }
19.
20.        if ($text !== 'Profile') {
21.            $url = LoginPage::$URL;
22.            $this->amOnPage($url);
23.            $this->waitPageLoad(10);
24.            $this->waitForElement('#username', 15);
25.            $this->fillField('#username', $username);
26.            $this->fillField('#password', $password);
27.            $this->click('_submit');
28.            $this->dontSee('Invalid credentials.');
29.        }
30.
31.        $this->comment('I am now logged');
32.    }
```

Výpis 6: Implementace funkce login

4.4 Vytvoření importu dat do nového systému skrze CSV soubory

V rámci praxe jsem také pracoval na projektu TONAK, což je e-shop nabízející produkty jak pro B2B tak i pro B2C zákazníky. Pro B2C klobouky, cylindry a barety a pro B2B kůže a látky, ze kterých jsou koncové produkty vyrobeny. E-shop musel tedy být rozdělen do části B2C a B2B kde se jednotlivé druhy produktu nabízejí danému zákazníkovi dle jeho typu. Mým úkolem na tomto projektu bylo naimportovat produkty obou druhů ze starého e-shopu do nového. Data ze starého e-shopu jsem dostal ve formátu CSV. Import nových dat byl komplikovaný zejména, protože nová struktura databáze neodpovídala staré, ze které se data exportovala a navíc množství importovaných dat bylo v řádech tisíců s velkým množství variant daných produktu musel se tedy klást velký důraz na optimalizaci importu pro co nejmenší zátěž na hosting daného e-shopu. Tuto problematiku jsem se rozhodl řešit tak, že všechny vazební tabulky k produktu si selectuji ještě před zahájením importu a v něm si budu ověřovat zda dané závislostí existuji, nebo zda potřebují vytvořit. Zda existuji tak ověřuji v PHP nikoliv samostatnými dotazy do databáze. Tím jsem omezil počet dotazů na databázi až o 70%. V rámci projektu jsme používali Doctrine 2 ORM. ORM umožnovalo s jednotlivými databázovými záznamy pracovat jako s objekty.



Obrázek 3: Databázová struktura produktů

Jeden z požadavků na import bylo, aby si jej mohl uživatel sám spouštět v administraci. V administraci jsem vytvořil formulář, kde uživatel vložil ZIP soubor, který obsahoval obrázky produktů a také CSV soubor s daty.
Import probíhal tak, že se nejdříve na základě nastavení v administraci rozhodlo, zda se jedná o produkty do části eshopu pro B2B nebo B2C. Načetl se ZIP soubor a vyjmuli se data. Nahráli se všechny obrázky na hosting a jejich jména se uložili do pole. Později se na základě jejich názvů přiřadili k jednotlivým produktům či variantám produktů pokud měl některý produkt referenci na neexitující obrázek uživatel o této skutečnosti byl informován prostřednictvím flash message a import selhal.

V další části importu se ověřilo, zda CSV soubor obsahoval všechny potřebné sloupce pro import. Toto ověření probíhalo tak, že se v poli specifikovaly názvy jednotlivých sloupců a vyjmul se první řádek z CSV importu, který obsahoval názvy sloupců a tyto názvy se také naplnily do pole. Pomocí cyklu foreach se iterovalo nad polem názvů a pokud PHP funkce in_array() nenalezla daný název, byl uživatel informován o chybějícím sloupci pomocí flash message a import selhal.

Po úspěšných ověřeních správnosti nahrávaných souborů začal samotný import. Import dat probíhal v cyklu while přičemž jedna iterace se rovnala jednomu řádku v CSV souboru.

Na začátku zpracování dat se muselo určit, zda daný řádek obsahuje informace o produktu a nebo o variantě nějakého produktu. Na základě toho se s daty pracovalo rozdílně. Následně se jednotlivá data plní do požadovaných objektů v rámci ORM. Pokud se k produktu či variantě přiřazuje nějaká závislost např. barva tak se nejdříve v předem naplněném poli selectem před zahájením importu zkontroluje, zda taková barva již existuje pokud ano pouze nastaví relaci v opačném případě barvu přídá, přiřadí referenci a poté aktualizuje pole s již existujícími barvami, což snižuje počet dotazů na databazi a zrychluje celý import.

5. Získané a využité zkušenosti a znalosti

Při vykonávání praxe jsem využil většinu zkušeností z programovacích předmětu nejvíce pak OOP. Nabyl jsem nových zkušeností s programovacím jazykem PHP. A blíže se seznámil s PHP frameworkem Symfony, koncept tohoto frameworku a práce s ním se mi velice zalíbila a chtěl bych na něm budovat další PHP aplikace i v budoucnu. Naučil jsem se pracovat s verzovacím nástrojem GIT, pracovat v týmu a aktivně komunikovat s členy týmu. Dále jsem využil také znalostí nabitých v rámci předmětu Mobilní zařízení, kde jsem zejména ocenil nabytých znalostí javascriptu, a kódování responzivního designu.

6. Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení

Během bakalářské praxe ve firmě Webvalley s. r. o. jsem nabyl mnoho nových znalostí o programovacím jazyku PHP a celkově vývoji webových aplikací. Všechny zadané úkoly jsem splnil s pomocí odborných rad a konzultací mých spolupracovníků. Díky absolvování této praxe jsem se setkal se skutečnými problémy na reálných projektech a dalo mi to určitý přehled o tom co se po PHP programátorech v praxi vyžaduje a co je dobré znát, popřípadě jakých zkušeností nabýt pro mé budoucí zaměstnání.

Literatura

1. Doctrine <https://www.doctrine-project.org/>
2. Git <https://git-scm.com/>
3. PHPUnit <https://phpunit.de/>
4. Webvalley <https://webvalley.cz/>
5. Codeception <https://codeception.com/>
6. Composer dokumentace <https://getcomposer.org/doc/>
7. Symfony <https://symfony.com/doc/current/index.html>